Compiling Knowledge-Based Systems to Ada: The PrkAda ProTalk Compiler

Robert E. Filman[*] and Paul H. Morris[†]
IntelliCorp, Inc.
1975 El Camino Real
Mountain View, California 94040
USA

Received 17 May 1996
Revised 9 October 1997

Abstract

Abstract This paper describes the implementation of the ProTalk compiler for PrkAda. An important component of the ProKappa system is its multi-paradigm, mixed-rule/imperative language, ProTalk. ProTalk combines elements of backtracking AI languages such as Prolog with imperative constructs such as conditionals, iteration, assignment, and subprogram invocation. Our implementation illustrates the use of generators to achieve a Prolog-like semantics, and the possibility (and difficulties) of employing such mechanisms in Ada.

*Keywords***:** Knowledge-based systems compilation, knowledge-based systems development tools, Ada, ProKappa, object-oriented systems, application delivery environments

1. Introduction

A fundamental premise of programming language design is that language constructs should capture intent. In AI, often the intent is to express quantified statements of the form of "when this happens, this should follow." Classical AI systems realize this paradigm with rule-based programming. (Rules are used frequently enough in AI that some confuse AI with rule-based programming.) However, rules alone are inadequate for the totality of systems building. Often, conventional imperatives such as sequencing, conditionals, iteration, assignment and subprogram invocation are the best expression of

[*] Current address: Advanced Technology Center, Lockheed Martin, 3251 Hanover Street O/H1–41 B/255, Palo Alto, California 94304, filman@ict.atc.lmco.com

[†] Current address: NASA AMES Research Center, M/S 269-1, Moffitt Field CA 94035, pmorris@ptolemy.arc.nasa.gov

design elements. This paper focuses on ProTalk, a language for ProKappa [1] that melds rule-based and imperative programming, and describes a compiler from ProTalk to Ada. The ProTalk-to-Ada compiler is a component of the PrkAda system, described in a companion paper [2].

Conceptually, pattern-action pairs are the basis for rule-based systems. When the pattern of a rule matches the situation, the rule's action may be executed. Rule languages are ambivalent about whether the pattern-actions express truth ("when this is true, conclude the following") or processes ("when this becomes true, do the following.")

Pattern-action languages have two important characteristics that distinguish them from conventional languages. The first is the need to describe patterns. This implies linguistic extension for pattern variables and pattern expressions, and a clear specification of the space of to-be-matched entities. The second centers on the issue of conflict resolution: what to do when several patterns match simultaneously [3]. The semantics of a particular pattern-action language may range from requiring this choice to be made non-deterministically through specifying complex rules for ordering the rule selection. Whatever point is chosen on this continuum, this conflict resolution problem introduces considerable intellectual complexity to the programming process.

Thus, rule languages are a double-edged sword. They allow "atomizing" knowledge (the independent declaration of separate facts), asserting universally quantified statements, and free the programmer from concern for control structures and sequencing. However, they present a non-deterministic semantics, introduce unanticipated interactions between elements of the atomized knowledge, require considerable effort in establishing context for each atomic knowledge element [4], and demand circumlocutions and idioms to obtain conventional control patterns such as loops and conditionals. Prominent examples of rule languages include Prolog [5] and OPS5 [6]. Almost every multi-paradigm AI tool has some form of a rule language.

Because rules have some of the semantics of if-then-else expressions, rules can be run (chained) either in a "forward" or "backward" direction. Forward chaining requires following all the consequences of an assertion. That is, if we have rules that state

$$\text{if } A \text{ then } B, \tag{1}$$

and

$$\text{if B then C}, \tag{2}$$

then a forward chaining system will, on the assertion of *A*, conclude *B* (using rule 1), then conclude *C* (using rule 2). OPS5 [7] and ART [8] are examples of forward chaining systems.

Backward chaining involves reasoning from a consequence to the conditions that cause that consequence. Thus, we could backward chain given the question "is *C* true?" to the question "is *B* true?" and then to the conclusion that *A's* truth implies *C's* truth. Prolog is an example of a backward chaining system. Some systems allow mixed chaining—it is possible to invoke forward chaining while back chaining, and vice versa. An example of this would be invoking back-chaining to satisfy one of the clauses in the condition of a forward-chaining rule. Of course, our "*A, B, C*" example simplifies the problem, as most rule systems allow variables for the patterns *A*, *B*, and *C*, and much of the search involves finding bindings for the pattern variables that satisfy the rules.

Rule systems also differ on whether they are *always active* or explicitly *invoked*. In always-active systems, rule-checking is immediately triggered by any assertion that is relevant to a condition of any rule. Many of these systems (such as RETE [7]) maintain a complex database of partial rule satisfactions in order to avoid redundantly rechecking

rule conditions. Invoked systems delay checking rules until the rule system is explicitly called. These systems generally allow search and rule-firing to be limited to specified rule sets.

There are four primary ways of implementing rule languages: interpreters, compilation to a network, compilation to an abstract machine, and compilation to generators. Interpretation (used, for example, in KEE [9]) requires an engine that successively examines the conditions of appropriate rules and explicitly executes a rule coherently. This requires little additional storage and allows selective invocations of subsets of all rules. However, it is not as directly efficient (in a raw-machine sense) as compilation mechanisms. The RETE mechanism compiles rules to a network, progressively advancing tokens through that network as it matches portions of rules. This is perhaps the most efficient way of implementing a collection of rules, but can have large space requirements, works directly only for forward-chaining, and does not explicitly limit the execution to rule subsets. Abstract machine compilation effectively treats rule languages as would a conventional compiler, determining a sequence of instructions that move data among registers. Abstract machines typically have primitives for pattern matching and storing the search context of the rule system. Compilation to the Warren Abstract Machine (WAM) is the standard strategy for implementing Prolog systems [10, 11]. Both continuation-based and generator-based search for Prolog have been explored by Horsell and Lander [12]. We discuss compilation by generators in Section 3, below.

ProTalk is a computer language that spans the continuum between conventional imperative languages and rule languages. That is, ProTalk lets the programmer use not only conventional programming constructs, but also pattern matching and rules. ProTalk in this respect traces its intellectual roots to Planner [13], which extended Lisp with pattern-matching and search constructs. Syntactically, ProTalk is marked by a (relatively) context-sensitive, “quasi-natural language” external form.

In the next section, we provide an overview of ProTalk, touching on its syntax, semantics, and interaction with the object system. We continue by discussing the ProTalk to Ada compiler, including the nature of backtracking, the use of tasks as generators in the compiler, and the compilation of the more difficult ProTalk constructs. We then summarize our results. A supplemental paper ([14]) provides both a more complete description of the ProTalk language; and an example of the translation of ProTalk to Ada.

2. ProTalk

ProTalk relies on the following fundamental concepts:

- **Rules and functions.** ProTalk subprograms are either functions (having parameters and returning values), or condition/action rules (both forward chaining and backward chaining). The bodies of both functions and rules employ the same basic language. The main difference is that functions are called directly, while rules use pattern-directed invocation.
- **Variables.** Variables in ProTalk are neither declared nor typed. (To put it another way, all values are of a single type, called a PrkType.) Variables are initially unbound. Like Prolog, evaluating an expression containing unbound variables may lead to bindings for those variables. However, unlike Prolog, one cannot unify two unbound variables, and, also unlike Prolog, one can reassign a new value to an already bound variable.
- **Success and failure.** Like Prolog, ProTalk incorporates a notion of success and failure. A ProTalk program is a series of statements. Statements include tests and simple unifications as well as assignments. Each statement either succeeds or fails. If a

statement succeeds, the system continues to the next statement. However, a statement may fail, for example, if a test evaluates to false, or if the system is unable to find an acceptable binding for a variable. Failure causes backtracking to the last choice point (where there were multiple ways of satisfying a condition, such as several possible bindings for a variable, or an explicit disjunction), and forces consideration of the next possibility. In the case of functions, only explicit choice points are allowed (created using the find operator). Rules have an implicit "find" before all statements. Thus, rules typically create many choice points. Unlike Prolog, in ProTalk one can also use conventional control constructs, (such as assignment, if/then/else, for/while, and iteration over lists and into accumulators) in functions and rules. In particular, the language provides a "return" from functions.

- **ProTalk and ProKappa**. ProTalk provides certain primitive operations needed for integration with the rest of ProKappa, for example, operations for changing/inquiring about class/member links, and slot and facet values in the object system.

2.1. A ProTalk Example

To go from the amorphously vague to the excruciatingly concrete, we consider the following example of a ProTalk function, Cs1ResetForRules, drawn from a reimplementation of CS1Fixer [15]. Cs1ResetForRules resets the state of the object system to prepare for the next problem-solving episode. In the code, the symbol "==" denotes a unification, which may succeed or fail, while "=" denotes an ordinary assignment, which always succeeds.

```
function Cs1ResetForRules (?self, ?slot)
                            /* Cs1ResetForRules is a method, so it
                             * takes two arguments, ?self, the object
                             * receiving the message, and ?slot, the
                             * name of the message.
                             */
{
    H2Source.Pressure = Normal;
                            /* Set the Pressure slot of the object named
                             * H2Source to the symbol Normal.
                             */

    for ?X in direct subclassof Subsystem;
                            /* For each immediate child of the class
                             * Subsystem, bind ?X to that child and
                            */
    do
    {
                            /* Find each value of the SubsystemProblem
                             * slot of ?X, binding it to ?sp.
                            */
        for {find ?X.SubsystemProblem      == ?sp; }
        do
        {
                            /* Remove the value ?sp from the
                             * SubsystemProblem of  ?X
```

```
                                */
                ?X.SubsystemProblem            -== ?sp;
            }
        }

        for ?Unit in instanceof Components;
                                /* For each instance descendant of
                                 * Components, do...
                                 */
        do
        {
                                /* If the instance is not in a specific list
                                 * of exceptions, then set its Value slot
                                 * to Normal
                                 */
           if Member (?Unit, '(E1@, E2@, E3@, E4@,
                               E5@, E6@, I1@, W1@))
                == FALSE;
           then
                ?Unit.Value = Normal;
        }

        for ?component =
                                /* Finish by running the rules in the set of
                                 * CoolingTestProcedureRules to find values
                                 * of the FaultyComponent slot of the
                                 * object Cs_1, printing out a message for.
                                * each
                                 */

          find [ CoolingTestProcedureRules ]
                           Cs_1.FaultyComponent;
            do {Print("\nA faulty component is ");
                Print(?component);
               }
    }
```

This function is intended to be called with ?self and ?slot bound. Calling a function with unbound variables is also possible. Such a function would act as a generator—successive calls, within the context of a search, produce new bindings for the unbound variables. When all bindings have been exhausted, the function call itself fails. (ProTalk requires that all parameters to a function must be bound by the time the function exits.)

The following is an example of a ProTalk rule—a backward chaining rule, to be precise.

```
        bcrule D1ReadsHighRule in AllLowAll_EsShortRules
                                /* The rule class for this rule is
                                 * AllLowAll_EsShortRules. (Rule classes
                                 * can themselves be built into directed
                                 * acyclic graphs, though (unlike KEE),
                                 * rules do not correspond to objects in the
```

```
                                * object system.)
                                */
{
  if:
   ?comp.Value  == High;
                                /* If some component, ?comp has High in
                                 * its Value slot, and
                                 */
   ?comp.Sensor == ?sen;
                                /* The value of its Sensor slot is ?sen, and
                                 */
   ?sen.Trend   == High;
                                /* That object, ?sen has High in its Trend
                                 * slot, then
                                 */
  then:
      ?comp.ComponentFailureType +== ReadsHigh;
                                /* Add ReadsHigh to the
                                 * ComponentFailure slot of ?comp
                                 */
      ?comp.FaultState            = UncorrectedFault;
                                /* Set its FaultState to UncorrectedFault
                                */
       RespondToHighSensor(?comp);
                                /* And call the function
                                 * RespondToHighSensor on ?comp.
                                 */
}
```

Since this is a backward chaining rule, it means, “If one is looking for an object with `ReadHigh` in its `ComponentFailureType` slot, or `UncorrectedFault` in its `FaultState` slot, then establish that the object has `High` in its `Value` slot, and its `sensor` slot contains another object that has `High` in its `Trend` slot.” If a rule succeeds, the rule itself is run, causing execution of the function `RespondToHighSensor` on the component.

3. ProTalk to Ada Compiler Design

The ProTalk compiler in the standard ProKappa system produces C output. Instead of developing a ProTalk/Ada compiler from scratch, we have modified the existing compiler so that it produces Ada code instead. This has several advantages. It works in much the same way, which is convenient for the user. From an implementation point of view, we saved considerable effort by reusing the existing parser mechanism. Only the code generation routines needed to be changed, and even there, much of the overall structure was preserved. This helps to maintain a close correspondence between ProTalk-C and ProTalk-Ada.

Because of important differences between C and Ada (primarily the lack of function pointers and dynamic long jumps in Ada[1]), the C-based approach for implementing ProTalk backtracking could not be directly implemented in Ada. It was necessary to develop a new code translation design. This is described in more detail in the following sections.

Since ProTalk functions support backtracking, and use the same language as rules, the latter provide little additional functionality beyond a sophisticated triggering mechanism. We have not implemented an Ada translator for rules, and will consider only the translation of functions during the remainder of this paper.

3.1. Backtracking in ProTalk

At a conceptual level, ProTalk behaves very much like Prolog, including mechanisms similar to the "cut" feature. However, ProTalk includes many additional high-level constructs to facilitate structured programming, readability, and ease of use.

A simplified ProTalk function has the form

```
function F {find S1; ... ; find Sn;}
```

A call to `F` may be viewed as a problem-solving episode. The function definition may be read as saying: "To solve `F`, find compatible solutions of subproblems S_1, ..., S_n and put them together to give a solution of `F`." The need for backtracking arises from the compatibility requirement. For example, it may turn out that a solution for S_1 is later discovered to be incompatible with subsequent function calls. (That is, there are no combinations of the bindings for variables in S_2, ..., S_n that succeed, given the choices made for variable bindings in S_1.) In that case, it is necessary to re-enter S_1 to find a new solution. This means that some record of the state of S_1's computation must be preserved so that the possible solutions will be available in an orderly fashion.

ProTalk provides mechanisms for controlling the amount of backtracking that occurs. If the "find" is omitted in front of a function call, then the call is *deterministic*. A deterministic call behaves more like a function call in a conventional language. It never fails when first entered. Moreover, if a deterministic call is re-entered during backtracking, a new solution is not found. Instead, the re-entry merely undoes any (local) effects of the call, and backtracking continues at the point preceding the call. (It should be noted that backtracking may occur *within* a deterministic call during the period it is first entered. It is only the re-entry backtracking and possible multiple solutions that are excluded by making the function deterministic.)

ProTalk also has a `find1` modifier. A call using `find1` is like a deterministic call except that it may also fail during the first entry. (Readers familiar with Prolog will observe that the behavior of `find1` resembles "cut.") Thus, `find1` requests at most one solution to a problem. There is, in addition, a more general `findn` modifier to specify a specific upper bound on the number of solutions to be computed.

[1] Ada, as used in this paper, refers to Ada 83. Ada 95 [16] has pointers to functions, allowing a more direct reimplementation of the original ProTalk compiler. This leaves open the question whether the Ada exception mechanism could be used to imitate the dynamic long jumps of C.

3.2. The ProTalk to C compiler

We give a brief description of the ProTalk-C approach so that the reader may better understand the changes required by the ProTalk-Ada design. The C compiler behaves as follows. Each ProTalk function F is implemented by several C functions, designated F_1, F_2, etc. We may think of these as computing the successive statements within F. However, the C functions do not simply find a solution to F and return. Instead, any pending ProTalk statements following a call to F are passed down to F_1 in an additional argument called the *continuation*. (This is more properly called a *success continuation*, since it describes what to do in the event of success in finding a solution. We consider *failure continuations* below.) The second and subsequent statements within F also get pushed onto the continuation after F_1 is called. As each statement in F succeeds and before the associated C function returns, the evaluator is called recursively on the next item in the continuation. Thus, successful subcalls do not return, but instead continue the computation below them until the continuation is finally exhausted.

More concretely, suppose F is defined in ProTalk as

```
function F {find G; find H;}
```

Then the C functions F_1, etc., are essentially defined as

```
  function F1(cont) { push(F2, cont); G(newcont); }
  function F2(cont) { push(F3, cont); H(newcont); }
function F3(cont) { Next = pop(cont); Next(newcont);}
```

Of course, G and H would also generally be implemented by several C-functions, so the subcalls would actually be to G1 and H1. If the statements in G all succeed, then F2 will be popped from the continuation and called, before G1 returns. This in turn may eventually lead to F3 being popped and called, and so on, all before G1 returns.

If the evaluation under G1 ultimately fails, then G1 does return, and control reverts to F1. If the definition of F had involved, say, a disjunction in the first statement, the F1 code would then work on an alternative solution for F. (Thus, the failure continuations are embedded in the C-code, rather than being passed as arguments.) When there are no more alternatives, we reach the end of the code for F1, and it returns, allowing its callers to try out their alternatives. Thus, we see that backtracking is implemented by means of simple C function returns.

If final success occurs, (i.e., if the success continuation list becomes empty), the C stack needs to be cleared. This is accomplished by a long jump back to the top-level evaluator call. Similarly, local successes of deterministic function calls require long jumps back to the point (on the C stack) where the deterministic call was entered.

The major difficulty in adapting this approach to Ada lies in the recursive evaluation of the continuation items. In C, function pointers can be treated as data and passed as arguments to other functions, where they can be subsequently evaluated. The continuation argument discussed above is represented as a data structure that includes function pointers. This is not allowed in Ada. Although it is possible to simulate the dynamic evaluation of “functional” data by means of case statements, this results in a loss of efficiency that would be particularly egregious for ProTalk because of the repeated need for such dynamic evaluation.

There were two practical disadvantages to following the existing C approach: the volume of additional runtime support code it requires, and the opacity of its generated code. The existing approach relies on a large, complex runtime system to reduce the

complexity of the compiler. Following that approach would have required reimplementing that system in Ada. The existing approach generates extremely opaque code. For example, the ProTalk variables are not directly translated as C variables, but must be packed into arrays and referenced indirectly by array location. (To echo the first point, a considerable volume of variable management code is required for purposes such as deassignment of variables on backtracking.) Similarly, the manipulation of continuations obscures the real work being done by the code.

3.3. Using Ada tasks for continuations

Fortunately, a more elegant design is available in Ada (which is, after all, a higher level language than C). C does not allow for saving a subprogram state between calls. Ada provides *tasks* with this capability.

In our approach, each ProTalk function becomes a task in Ada. The points where the function is first entered, and re-entered during backtracking correspond to Ada task entries. (The state of an Ada task survives between entry calls.) The Ada task terminates when either backtracking exhausts all the solutions for the function, or final success occurs for the top-level call or an intermediate deterministic call.

The F example translates into an Ada task essentially as follows.

```
task body F_task is
begin
    accept start;

    loop
        Call(G_task); exit when Fails;
        loop
            Call(H_task); exit when Fails;
            Success;
            accept stop;
            accept start;
        end loop;
    end loop;

    Failure;
    accept stop;
end F_task;
```

Note that there are two entries, start and stop. The procedures Success and Failure set global variables that communicate the status of the computation. The nested loops ensure that all possible solutions for G and H are tried if necessary. That is, they help implement the backtracking. The F task is called as follows.

```
procedure Call(F_task) is
begin
    F_task.start;
    F_task.stop;
end Call;
```

If the ProTalk functions have variables, these are carried over directly as arguments to the start/stop entries. Thus, there is a direct textual correspondence to the original ProTalk code. Moreover, there is no need for a success-continuation argument.

With this approach, the bulk of the work involved designing and implementing individualized translations for the specialized ProTalk statements such as assignments, if-statements, or-statements, for-loops, and so on. The runtime support requirements are relatively trivial, with the major work being done by the compiler.

3.4. ProTalk statement translations

We now discuss the translations of the different statement types. Table 1. lists the statement types available in ProTalk. Translation of many of these statement types is straightforward, and differs little from the mechanism of the ProTalk-to-C compiler (except, of course, for the vagaries of target syntax). For example, an assignment statement in ProTalk maps to an assignment statement in Ada. We confine our discussion to those that are particularly relevant to the task-oriented design. The statement types of main interest fall into three groups: conditional branching statements (OR, IF, NOT); iterative looping statements (FOR, WHILE); and the RETURN statement.

Table 1. ProTalk Statement Types

succeed	fail	proccall	cfn
ccode	return	findn	accum
retry	for	while	assign
inarray	inlist	fromto	tuple
test	and	bound	binrel
findany	or	if	not

3.4.1. Conditional branching statements

The reader is reminded that ProTalk's backtracking produces a nonstandard semantics for statements. Most importantly, every statement is a conditional because its success or failure governs the applicability of subsequent statements. For example, the ProTalk statement

```
NOT S;
```

is equivalent to

```
if S; then FAIL; else SUCCEED;
```

If the first branch applies, backtracking begins immediately; otherwise the subsequent statements are evaluated.

The most general cases of IF and OR are implemented as subtasks in ProTalk-Ada. The following examples illustrate the translations. (This is a simplification that glosses over possible structure introduced by the substatements within the OR.)

The statement `find OR {S1; S2; S3;}` generates a subtask defined by:

```
task body OR_task is begin
    accept start;

    for each solution of S1 loop
        success;
        accept stop;
```

```
        accept start;
    end loop;

    for each solution of S2 loop
        success;
        accept stop;
        accept start;
    end loop;

    for each solution of S3 loop
        success;
        accept stop;
        accept start;
    end loop;

    failure;
    accept stop;
end OR_task;
```

The subtask is called within the main body of the code as follows:

```
...
loop
    Call(OR_task);
    exit when fails;
    ...
end loop;
```

The behavior of the IF construct is somewhat complex with respect to backtracking. The conditional part of the IF is evaluated to see if it has a solution. During this evaluation, backtracking may occur within the conditional part. However, if the evaluation succeeds, and the computation has proceeded to the THEN part, no further solutions to the conditional part are sought. That is, if the THEN part subsequently fails, then the whole IF fails. More precisely, the following (generic) IF statement:

```
IF C1; THEN S1; ELSE IF C2; THEN S2;
```

is semantically equivalent to the following two ProTalk statements, which use an auxiliary variable BRANCH:

```
find1 OR
    {
        {C1;   BRANCH == 1;}     /* This sets BRANCH */
        {true; BRANCH == 2;}
    }
find OR
    {
        {BRANCH == 1; S1;}       /* This tests BRANCH */
        {BRANCH == 2; S2;}
    }
```

However, instead of following this approach, the Ada translation of IF makes backtracking jump over the conditional part by raising an Ada exception. Thus, the statement

```
IF C1; THEN S1; ELSE S2;
```

generates a subtask defined by:

```
task body IF_task is begin
    accept start;

    declare
        IF_exit : exception;
    begin
        for each solution of C1 loop
            for each solution of S1 loop
                success;
                accept stop;
                accept start;
            end loop;
            raise IF_exit;
        end loop;

        for each solution of S2 loop
            success;
            accept stop;
            accept start;
        end loop;
    exception
        when IF_exit => null;
    end;

    failure;
    accept stop;
end IF_task;
```

3.4.2. Return statement

A return statement indicates an early successful return from a function. If the return occurs as a top level statement, this is easily implemented: the compiler simply omits any subsequent statements and goes directly to the success exit of the task.

The situation is more complicated when the return is nested within a conditional branching statement, because there may be additional statements after the conditional statement. The additional statements should only be executed if a branch without the return is chosen.

This complication is handled by using distributivity to absorb any following statements inside each branch of the conditional statement. To facilitate this, a new subtask, called a continuation subtask, is formed to encapsulate all the subsequent statements, and calls to this subtask are placed at the end of each branch of the conditional. Thus,

```
...
find or {S1; S2;}
continuation;
```

becomes

```
...
```

```
find or
  {
    {S1; continuation;}
    {S2; continuation;}
  }
```

Now any branches that contain a return can handle it the same way as if it occurred at the top level: statements following the return (including the continuation) are simply deleted.

3.4.3. Iterative statements

The FOR and WHILE statements are generally handled similarly to ProTalk-C. We note that `FOR S1; DO S2;` is compiled equivalently to `NOT {S1; S2; FAIL;}`. The WHILE statement is like FOR but always reevaluates its conditional from the beginning. More precisely, `WHILE S1; DO S2;` is equivalent to `FOR RETRY S1; DO S2;` where `RETRY S;` is like `FIND1 S;` except that instead of failing on backtracking, it repeatedly regenerates the first solution for `S`.

These generally compile into simple loops in Ada. The only complication that arises is when a RETURN statement is nested within the FOR/WHILE. In this situation, we wrap a subtask around the iteration, and then handle it in a manner similar to that described above for conditional branching statements. That is, the statements following the iteration are folded into a continuation subtask. The continuation subtask is then called after the iteration. The overall structure for a `FOR S1; DO S2;` that contains an embedded return looks as follows:

```
task body FOR_task is begin
    accept start;

    for each solution of S1 loop
        for each solution of S2 loop
            null;
        end loop;
    end loop;

    for each solution of the continuation-subtask loop
        success;
        accept stop;
        accept start;
    end loop;

    failure;
    accept stop;
end FOR_task;
```

It should be noted that any embedded return in S1 or S2 will generate a success exit from the FOR task, bypassing the continuation subtask.

4. Concluding Remarks

The preceding describes the main techniques used to compile ProTalk constructs into Ada using tasks. There are further complications that we have not discussed which result from the need to propagate Ada exceptions through the tasks, and to do memory management

(garbage collection). These require additional bookkeeping code in the Ada tasks but do not otherwise affect the translations. Unfortunately, the additions again have the effect of obscuring the underlying structure of the code, though in a different way than in the C-based translation. (Note that the latter does not need to compile explicit memory management instructions into the translation because C provides sufficient facilities to build a runtime memory management system.)

The ProTalk to Ada compiler illustrates the possibilities (and difficulties) of building AI applications in Ada. Ada restrictions, such as the lack of functional values, make conventional, abstract-machine approaches inappropriate. On the other hand, Ada's tasking mechanism allows an alternative (and, in many ways, clearer) solution that uses task-based generators to retain problem-state information.

Acknowledgments

ProTalk is a component of ProKappa, an object-oriented software development environment marketed by IntelliCorp. It is difficult at this stage to trace the precise contributions of various individuals in its development. It is related to early design work on "Assertion Nets" by Conrad Bock [17]. The standard ProTalk-to-C compiler that is part of ProKappa was written by Greg Clemenson. The contribution of the present authors is limited to work on the design and implementation of the translator to Ada.

This work was performed while the authors were working for IntelliCorp, Inc, and supported by NASA/Marshall Space Flight Center under contract NAS 8-38488.

References

[1] IntelliCorp, Inc., *ProKappa Reference Manuals*, Pub. No. PK2.0–RM1–2, (1991).

[2] R. E. Filman and P. H. Morris, *Compiling Knowledge-Based Systems to Ada: The PrkAda Core*, International Journal on Artificial Intelligence Tools, this issue, (1997).

[3] R. Davis and J. King, *An overview of production systems*, Machine Intelligence 8, eds. E. W. Elcock and D. Michie, Wiley, New York (1976) 300–332.

[4] J. Bachant and E. Soloway, *The Engineering of XCON*, Comm. ACM **32** (1989) 310–319.

[5] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer Verlag, New York (1984).

[6] L. Brownston, R. Farrell, E. Kant and N. Martin, *Programming Expert Systems in OPS5*, Addison-Wesley, Reading, Massachusetts (1985).

[7] C. L. Forgy, *Rete: A fast algorithm for the many pattern/many object matching problem*, Artificial Intelligence **19** (1982) 17–37.

[8] C. Williams, *ART The Advanced Reasoning Tool—Conceptual Overview*, Inference Corp., Los Angeles (1984).

[9] R. E. Fikes and T. Kehler, *The role of frame-based representation in reasoning*, CACM, **28** (1985) 904–920.

[10] D. H. D. Warren, *Implementing Prolog—Compiling Predicate Logic Programs*, Univ. of Edinburgh D.A.I. Research Report 39–40 (1977).

[11] D. H. D. Warren, *An Abstract Prolog Instruction Set*, Technical report number 309, Artificial Intelligence Center, SRI International (1983).

[12] R. K. Horsell and L. C. Lander, *Logic Programming in Ada*, Sixth Annual Conference on Artificial Intelligence and Ada, eds. J. Baldo, J. Diaz-Herrera, and D. Littman, Reston, VA (Nov. 1990) 122–133

[13] C. Hewitt, *Description and theoretical analysis (using schemata) of PLANNER: A language for proving theorems and manipulating models in a robot*. Doct. dissertation, MIT (1971).

[14] R. E. Filman and P. H. Morris, *Implementation Notes for PrkAda*, http://www.best.com/~morris/ftp/prkimp, (1997).
[15] J. T. Malin and N. Lance, *Processes in construction of failure management expert systems from device design information*, IEEE Trans. on Sys., Man and Cyber. **SMC-17** (1987) 956–967.
[16] Ada 95 Mapping/Revision Team, *Programming Language Ada: Language and Standard Libraries*, ISO/IEC DIS 8652, Intermetrics, Cambridge, Massachusetts, (1994).
[17] C. Bock, R. Filman, P. Morris and R. Treitel, *Next-generation knowledge system tools*, Proc. AAAI Symp. on Knowledge System Development Tools and Languages, Stanford, California (1989) 1—5.